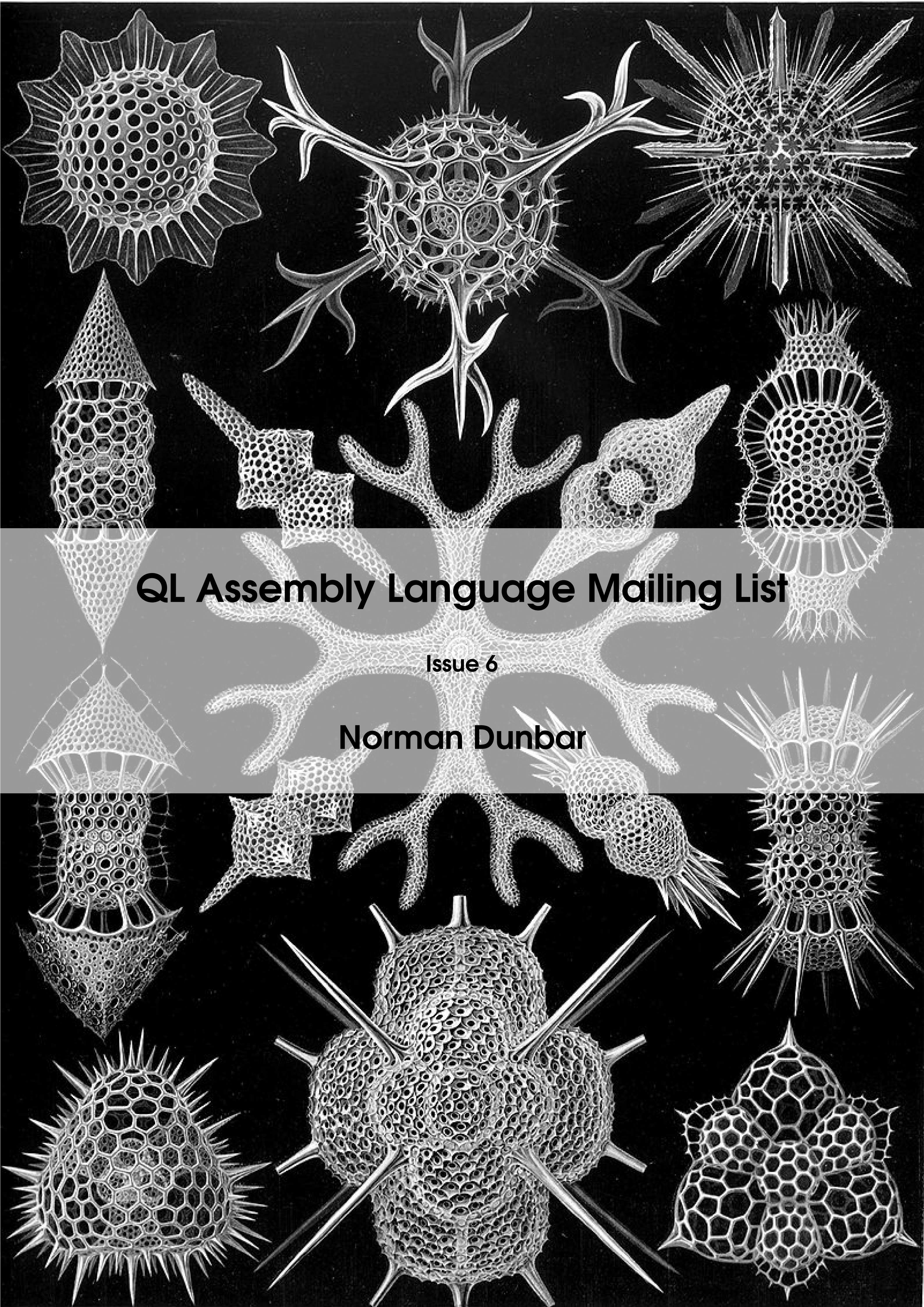


QL Assembly Language Mailing List

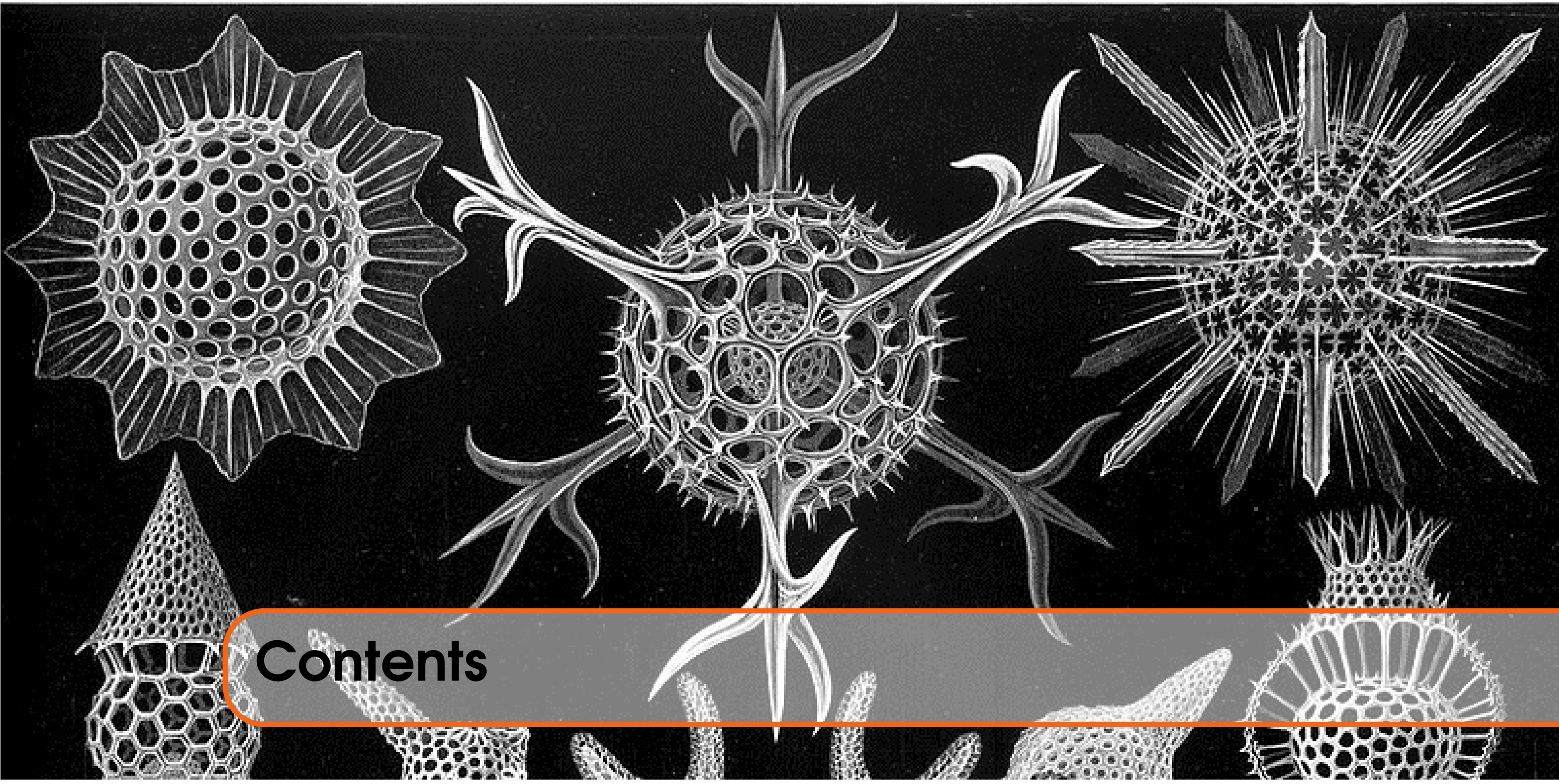
Issue 6

Norman Dunbar

PUBLISHED BY MEMYSELFEYE PUBLISHING ;-)

Download from:
https://github.com/NormanDunbar/QLAssemblyLanguageMagazine/blob/Issue_006/Issue_006/Assembly_Language_006.pdf

This pdf document was created on *15/12/2018* at *14:34:38*.



Contents

1 Preface 7

1.1 Feedback 7

1.2 Subscribing to The Mailing List 7

1.3 Contacting The Mailing List 8

2 Feedback on Issue 5 9

2.1 No Feedback so far! 9

3 Cross Compiled Programs 11

3.0.1 The XTcc Trailer Record 11

3.0.2 Program Description 12

3.0.3 The Program Listing 12

4 Using the MC68020 - Part 3 19

4.1 Status Register 19

4.1.1 Trace Bits T_1 and T_0 19

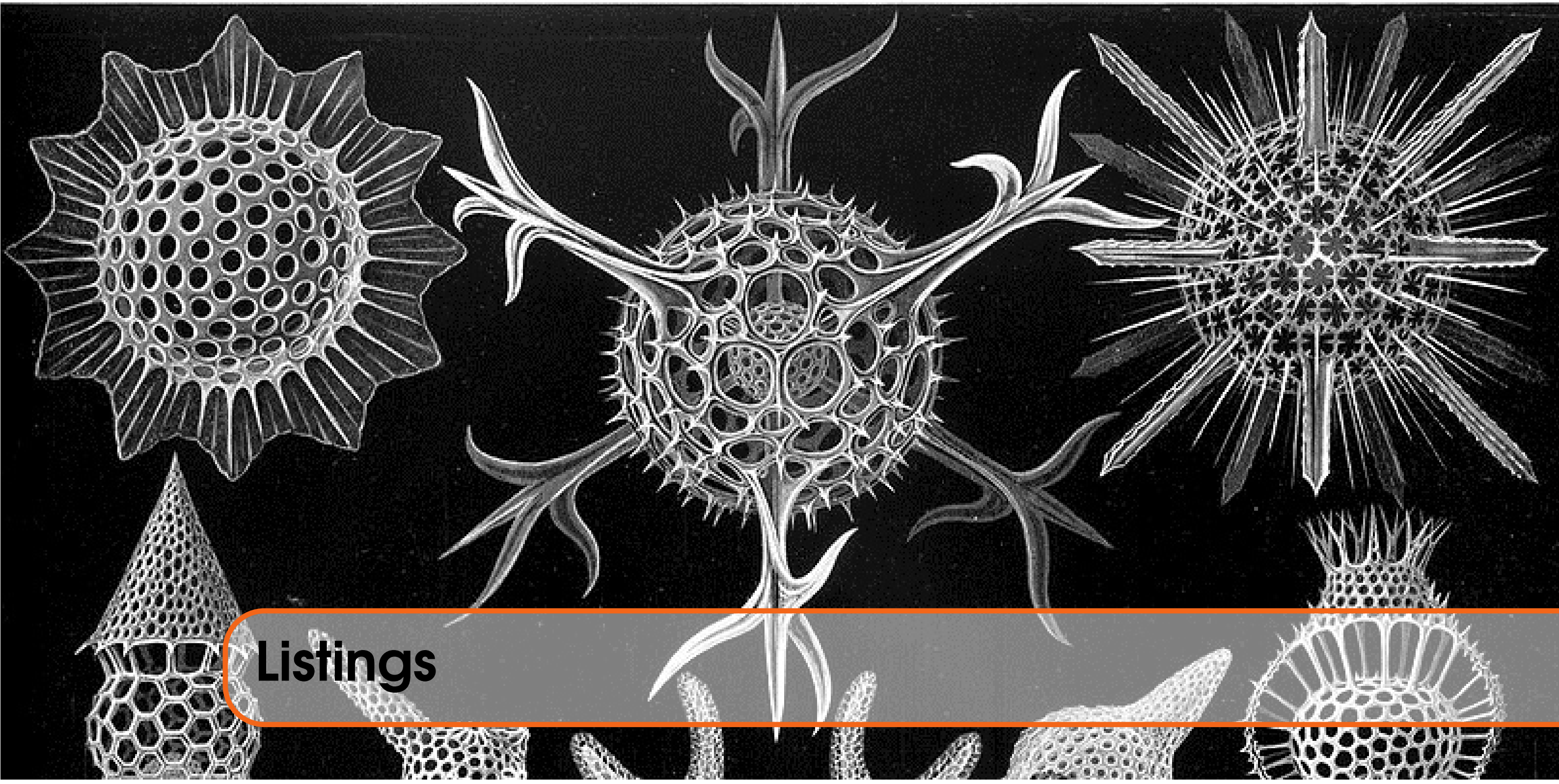
4.1.2 Supervisor Master and Interrupt Modes 20

4.2 Control Registers and `MOVEC` 20

4.2.1 SFC and DFC- Source and Destination Function Code 20

4.2.2 VBR - vector Base Register 21
4.2.3 CACR and CAAR - Cache Control 21
4.2.4 USP, MSP and ISP - Stack Pointers 21

5 Image Credits 23



Listings

3.1 XTcc - Comments 12
3.2 XTcc - Equates 12
3.3 XTcc - Job Start 13
3.4 XTcc - Channel Checking 14
3.5 XTcc - Read the File Header 14
3.6 XTcc - Is the File Executable? 15
3.7 XTcc - Locating the XTcc Trailer 15
3.8 XTcc - Read the XTcc Trailer Record 15
3.9 XTcc - Setting the Header Data 16
3.10 XTcc - Writing the Header 17
3.11 XTcc - Termination 17

1. Preface

1.1 Feedback

Please send all feedback to assembly@qdosmsq.dunbar-it.co.uk. You may also send articles to this address, however, please note that anything sent to this email address may be used in a future issue of the eMagazine. Please mark your email clearly if you do not wish this to happen.

This eMagazine is created in LaTeXsource format, aka plain text with a few formatting commands thrown in for good measure, so I can cope with almost any format you might want to send me. As long as I can get plain text out of it, I can convert it to a suitable source format with reasonable ease.

I use a Linux system to generate this eMagazine so I can read most, if not all, Word or MS Office documents, Quill, Plain text, email etc formats. Text87 might be a problem though!

1.2 Subscribing to The Mailing List

This eMagazine is available by subscribing to the mailing list. You do this by sending your favourite browser to http://qdosmsq.dunbar-it.co.uk/mailinglist and clicking on the link “Subscribe to our Newsletters”.

On the next screen, you are invited to enter your email address *twice*, and your name. If you wish to receive emails from the mailing list in HTML format then tick the box that offers you that option. Click the Subscribe button.

An email will be sent to you with a link that you must click on to confirm your subscription. Once done, that is all you need to do. The rest is up to me!

1.3 Contacting The Mailing List

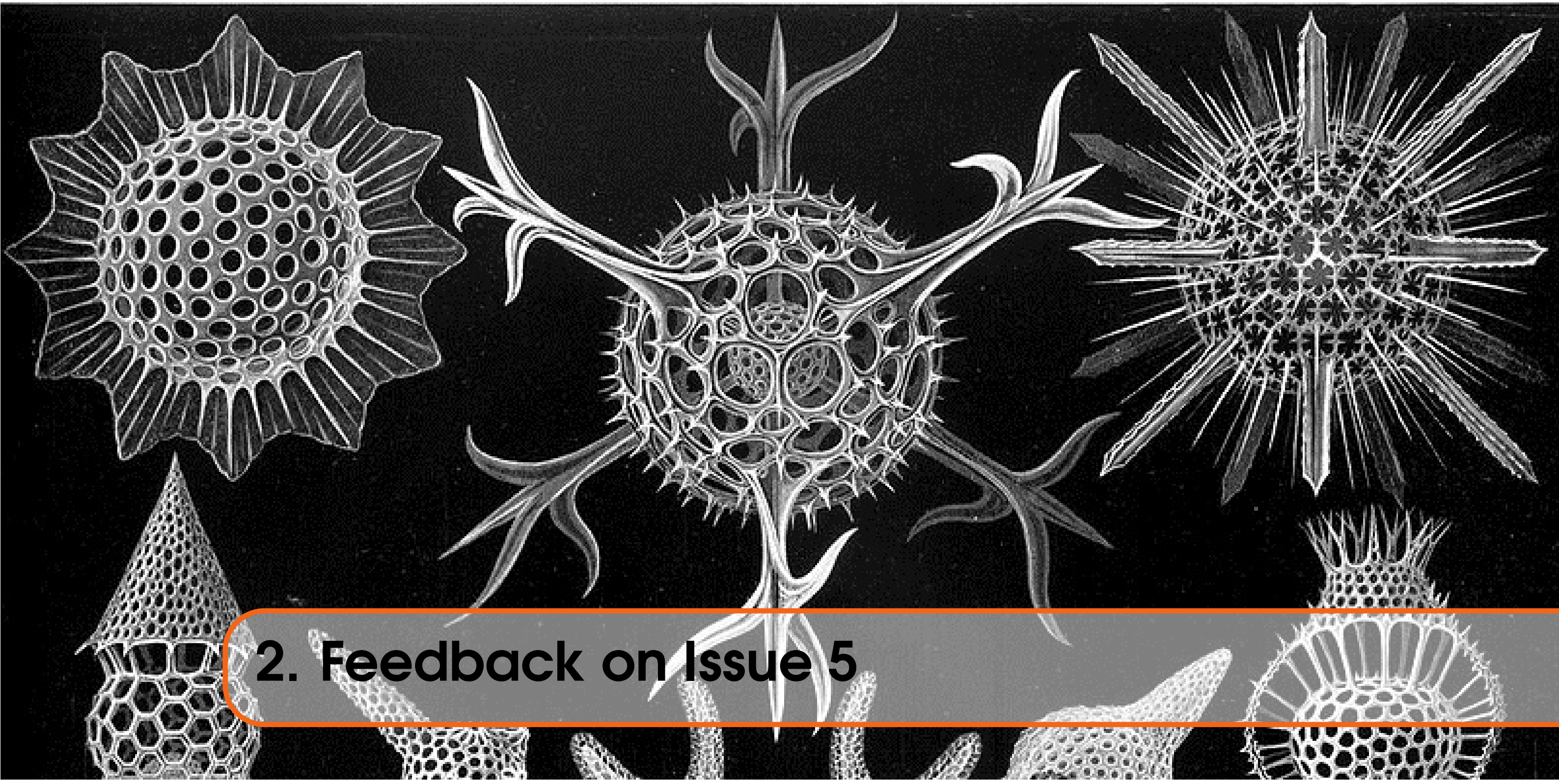
I'm rather hoping that this mailing list will not be a one-way affair, like QL Today appeared to be. I'm very open to suggestions, opinions, articles etc from my readers, otherwise how do I know what I'm doing is right or wrong?

I suspect George will continue to keep me correct on matters where I get stuff completely wrong, as before, and I know George did ask if the list would be contactable, so I've set up an email address for the list, so that you can make comments etc as you wish. The email address is:

`assembly@qdosmsq.dunbar-it.co.uk`

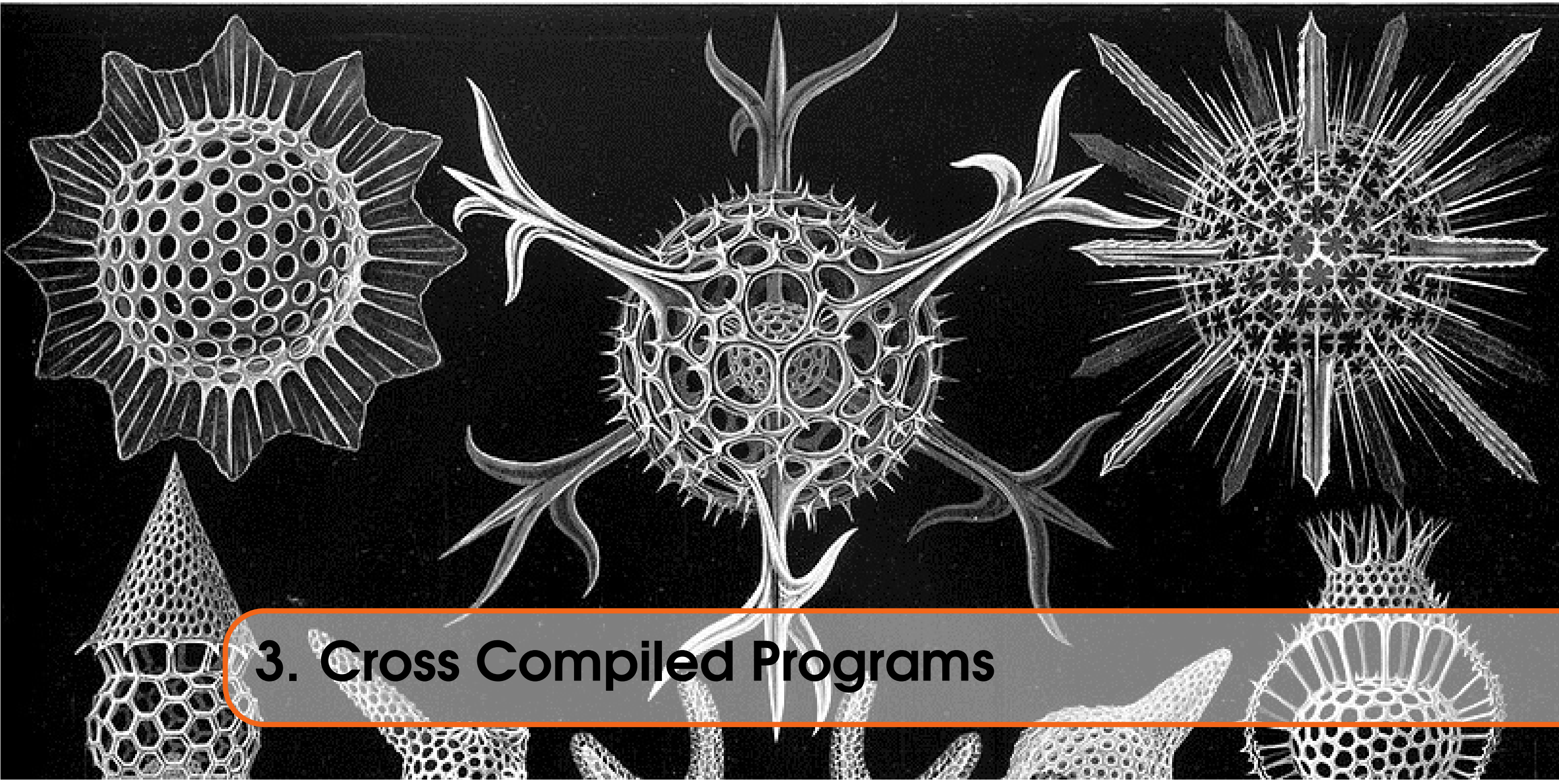
Any emails sent there will eventually find me. Please note, anything sent to that email address will be considered for publication, so I would appreciate your name at the very least if you intend to send something. If you do not wish your email to be considered for publication, please mark it clearly as such, thanks. I look forward to hearing from you all, from time to time.

If you do have an article to contribute, I'll happily accept it in almost any format - email, text, Word, Libre/Open Office odt, Quill, PC Quill, etc etc. Ideally, a LaTeXsource document is the best format, because I can simply include those directly, but I doubt I'll be getting many of those! But not to worry, if you have something, I'll hopefully manage to include it.



2. Feedback on Issue 5

2.1 No Feedback so far!



3. Cross Compiled Programs

Recently, I've been playing about with the xtc68 C compiler - which is basically C68 for Linux (or Windows, if you must!) and allows me to have fun writing C68 programs on my Linux laptop, which will be eventually copied over to the QL, and executed there.

As ever, any computer that is *not* a QL (or an emulator) has a problem when executable files are involved - there's no file header present, so there's no easy way to make the file executable on the QL - other than making up some number for the data space, allocating a chunk of RAM equal to the file size, loading it into that RAM area with LBYTES and then SEXECing the file back to the device. There has to be an easier way, surely?

I started a thread on QLForum about this cross compiler, and somewhere in that thread, I put up the code for a SuperBasic utility to fix up the dataspace for these compiled files. The forum thread is at `https://qlforum.co.uk/viewtopic.php?f=3&t=2605`. However, it wasn't quite what I really needed, plus, I couldn't really write an article for the eComic if the code was in SuperBASIC, could I?

Step forward my XTcc utility, described later. This utility does all the needful to get a file on the QL from its unusable state to a executable - very handy for files compiled with the xtc68 compiler or anything else that writes an XTcc trailer to the compiled file. I only know of the xtc68 compiler which does this, but there may be others. (Feedback very welcome.)

3.0.1 The XTcc Trailer Record

The trailer record produced by the compiler, and any other applications that create it, is a simple addition of 8 bytes to the very end of the file in question. These 8 bytes are split into two 4 byte chunks:

- The text "XTcc" in exactly that letter case.
- The required data space for the QL file, in big endian, long word format.

3.0.2 Program Description

The program, XTcc, is quite simple and carries out the following steps after being executed as a filter:

- Checks that only one filename was supplied, exits with a Bad Parameter error if not.
- Reads the file's header.
- If the file is already an executable file, then exits quietly as there is nothing more to do.
- Reads the file's length from the header, and sets the file pointer to that position minus 8 bytes. If the file cannot be positions at the required place, exit with an Out of Range error.
- Reads the last 8 bytes of the file. Exits with a File Error if 8 bytes couldn't be read.
- Checks that the first 4 bytes read are "XTcc", if not, exits with a Not Found error.
- Copies the data space from the last 4 bytes of the file into the file header.
- Sets the file's type, in the header, to be executable.
- Writes the file header back to the medium.
- The job then exits as if nothing had happened.

3.0.3 The Program Listing

```
;------------------------------------------------------------------------------
; XTcc:
;
; This utility reads a cross-compiled executable for QDOSMSQ and will
; attempt to correctly set the file's data space according to the
; 'XTcc' setting stored at the end of the file.
;
;
; EX XTcc_bin, input_file
;
;------------------------------------------------------------------------------
; 13/12/2018 NDunbar Created for QDOSMSQ Assembly Mailing List.
;------------------------------------------------------------------------------
; (c) Norman Dunbar, 2018. Permission granted for unlimited use
; or abuse, without attribution being required. Just enjoy!
;------------------------------------------------------------------------------
```

Listing 3.1: XTcc - Comments

Nothing to see here except some blurb explaining what the code is for and how to execute the utility.

```

; How many channels do I want?
NUMCHANS
            equ     1                   ; How many channels required?


; Stack stuff.
sourceId
            equ     $02                 ; Offset(A7) to input file id

; Other stuff.
err_nc
            equ     -1                  ; Not complete.
err_or
```

```
                equ     -4                          ; Out of range.
err_nf
                equ     -7                          ; Not found.
err_bp
                equ     -15                         ; Bad parameter.
err_fe
                equ     -16                         ; File error.
timeout
                equ     -1                          ; Trap call timeouts.
me
                equ     -1                          ; Job id for this job.
exeType
                equ     $01                         ; File Type for executable.
fileType
                equ     $05                         ; Offset in header to file type.
fileSize
                equ     $00                         ; Offset to file length.
fileData
                equ     $06                         ; Offset to dataspace in header
```

Listing 3.2: XTcc - Equates

The code above simply initialises various equates that will be required elsewhere.

```

;======================================================================
; Here begins the code.
;----------------------------------------------------------------------
; Stack on entry:
;
; $0c(a7) = bytes of parameter + padding, if odd length.
; $0a(a7) = Parameter size word.
; $06(a7) = Output file channel id.
; $02(a7) = Source file channel id.
; $00(a7) = How many channels? Should be $02.
;======================================================================
start
                bra.s   checkStack

                dc.l    $00
                dc.w    $4afb
name
                dc.w    name_end-name-2
                dc.b    'XTcc'
name_end
                equ     *

version
                dc.w    vers_end-version-2
                dc.b    'Version 1.00 - 13/Dec/2018'
vers_end
                equ     *

rh_buffer
                ds.w    32                          ; Storage for file header
xtcc_buffer
                ds.l    2                           ; Storage for XTcc flag *
```

```
    ⟹ dataspace
```

Listing 3.3: XTcc - Job Start

Now we are getting interesting. The start of the code is as above, and it consists of the standard QDOSMSQ job header followed by a version number for the utility - which is, currently, unused in the remainder of the code - followed by the defining of two buffers. One buffer is 64 bytes long for the file header and the other is 8 for the XTcc Trailer Record data.

```

;------------------------------------------------------------------------------
; Check the stack on entry. We only require NUMCHAN channels - any
; thing other than NUMCHANS will result in a BAD PARAMETER error on
; exit from EW (but sadly, not from EX).
;------------------------------------------------------------------------------
checkStack
            cmpi.w   #NUMCHANS,(a7)        ; One channel is a must
            beq.s    readHeader            ; Ok
            moveq    #err_bp,d0            ; Oops
            bra.s    errorExit             ; Bale out
```

Listing 3.4: XTcc - Channel Checking

The first check made by the code is to ensure that it was called with a single file channel on the stack. The utility wil exit with a bad parameter error if this is not the case.

```

;------------------------------------------------------------------------------
; READ_HEADER = read the file header for the given channel.
;
; A0.L = Channel Id.          (Preserved)
; A1.L = Buffer address.      (1 past end of buffer on return)
; D1   = Not used.            (Size of buffer read)
; D2.W = Buffer length.       (Preserved)
; D3.W = Timeout.             (Preserved)
;------------------------------------------------------------------------------
readHeader
            moveq    #fs_headr,d0          ; Reading the header
            moveq    #64,d2                ; Buffer maximum size
            moveq    #timeout,d3           ; Infinity is preserved
    ⟹ throughout
            move.l   sourceId(a7),a0       ; Input channel ID - preserved
            lea      rh_buffer,a1          ; Header buffer address
            move.l   a1,a3                 ; Preserve buffer address
            move.w   #64,d2                ; Buffer maximum length
            trap     #3                    ; Do it
            tst.l    d0                    ; Check errors
            bne.s    errorExit             ; Oh dear!
            cmp.w    d1,d2                 ; Successful read?
            beq.s    checkExecutableType   ; Yes
            moveq    #err_nc,d0            ; Not Complete
            bra.s    errorExit             ; Depart
```

Listing 3.5: XTcc - Read the File Header

Reading the passed file's header is next. There should be 64 bytes to be read and this is checked on return form the trap. If we didn't get exactly 64 bytes, we bale out with a not complete error.

Interestingly, I noticed that in QPC version 4.0.5, if the file was ever renamed, the file header appears to retain the original name. That caused me no end of *fun*[1] when I was debugging - reading the header for one file, and getting a completely different file's header, or so it seemed.

```

;------------------------------------------------------------------------------
; Check if the file is already executable. If so, quietly exit as we
; have nothing to do. Cross compiled files do not come set to be
; executable.
;------------------------------------------------------------------------------
checkExecutableType
            cmpi.b  #exeType,fileType(a3)  ; Buffer start is in a3 now
            beq.s   allDone                ; Executable - nothing to do
```

Listing 3.6: XTcc - Is the File Executable?

If the header was happily read, the code above makes sure that the file's type is not already executable. If it is, the utility will simply exit as there is nothing more to do. Cross compiled files don't come with the file's type set to executable.

```

;------------------------------------------------------------------------------
; In a cross compiled file, there is a pair of long words at the very
; end of the file. These are 'XTcc' followed by the data space for
; QDOSMSQ.
;------------------------------------------------------------------------------
; FS_POSAB:
;
; A0.L = Channel Id.          (Preserved)
; A1.L = Not used.            (Corrupted!)
; D1.L = File position.       (New file position on return)
; D3.W = Timeout.             (Preserved)
;------------------------------------------------------------------------------
setFileToXTcc
            moveq   #fs_posab,d0           ; Position absolutely
            move.l  fileSize(a3),d1        ; Get file size
            subq.l  #8,d1                  ; Point at XTcc location in file
            move.l  d1,d2                  ; Save required position
            trap    #3                     ; Do it
            tst.l   d0                     ; Ok?
            bne.s   errorExit              ; Oops!
            cmp.l   d1,d2                  ; Actual = requested position?
            beq.s   readXTccData           ; Yes
            moveq   #err_or,d0             ; Out of range
            bra.s   errorExit              ; Bale out
```

Listing 3.7: XTcc - Locating the XTcc Trailer

The header was read and the file isn't executable. The next step is to position the file's read pointer at 8 bytes back from the very end of the file. This is where we expect to find the XTcc Trailer Record that we need. If we fail to set the position exactly as requested, we bale out with an out of range error.

```

;------------------------------------------------------------------------------
; Read the final 2 words from the input file.
```

[1] For certain values of 'fun'!

```
;------------------------------------------------------------------------------
; IO_FSTRG:
;
; A0.L = Channel Id.          (Preserved)
; A1.L = Buffer address.      (Old A1 + returned D1.W)
; D1.L = Not Used.            (Number of bytes read)
; D2.W = Buffer size.         (Preserved)
; D3.W = Timeout.             (Preserved)
;------------------------------------------------------------------------------
readXTccData
                moveq   #io_fstrg,d0            ; Fetch bytes
                moveq   #8,d2                   ; Bytes we want
                lea     xtcc_buffer,a1          ; Buffer address
                move.l  a1,a2                   ; Save buffer address
                trap    #3                      ; Do it
                tst.l   d0
                bne.s   errorExit               ; Oops!
                cmp.w   d2,d1                   ; Did we get 8 bytes?
                beq.s   checkXTccFound          ; Yes
                moveq   #err_fe,d0              ; -16 File Error
                bra.s   errorExit               ; Bale out
```

Listing 3.8: XTcc - Read the XTcc Trailer Record

Next up, we read the 8 bytes that make up the XTcc Trailer Record. If this fails, or we do not read exactly 8 bytes, bale out with a file error message.

```

;------------------------------------------------------------------------------
; We should have 'XTcc' in the buffer plus the dataspace required.
;------------------------------------------------------------------------------
checkXTccFound
                cmpi.l  #"XTcc",(a2)+           ; Got the flag?
                bne.s   noXTccFound             ; Nope

;------------------------------------------------------------------------------
; We have the data we want, copy the dataspace into the file header
; and then make the file executable.
;------------------------------------------------------------------------------
extractDataSpace
                move.l  (a2),fileData(a3)       ; Copy the value over
                move.b  #exeType,fileType(a3)           ; Make executable
                bra.s   writeHeader             ; Write the header back

;------------------------------------------------------------------------------
; We didn't find the "XTcc" flag at the end of the file.
;------------------------------------------------------------------------------
noXTccFound
                moveq   #err_nf,d0              ; Not found
                bra.s   errorExit               ; Bale out
```

Listing 3.9: XTcc - Setting the Header Data

Assuming that we managed to read it, does the XTcc Trailer start with the XTcc flag, which happens to be the string "XTcc" in that letter case. In the event that we didn't find that flag, we will exit with a not found error.

If the flag is found, copy the last 4 bytes of the XTcc Trailer into the file's header to set the data space, and set the file's type to be an executable file.

```

;------------------------------------------------------------------------
; Write the file header for the given channel.
;
; A0.L = Channel Id.         (Preserved)
; A1.L = Buffer address.     (Corrupted)
; D1   = Not used.           (Length of set header)
; D2   = Not used.           (Preserved)
; D3.W = Timeout.            (Preserved)
;------------------------------------------------------------------------
writeHeader
                moveq    #fs_heads,d0          ; Write the header
                move.l   a3,a1                 ; Header buffer
                trap     #3                    ; Do it
                tst.l    d0                    ; Ok?
                bne.s    errorExit             ; Sadly, not!
```

Listing 3.10: XTcc - Writing the Header

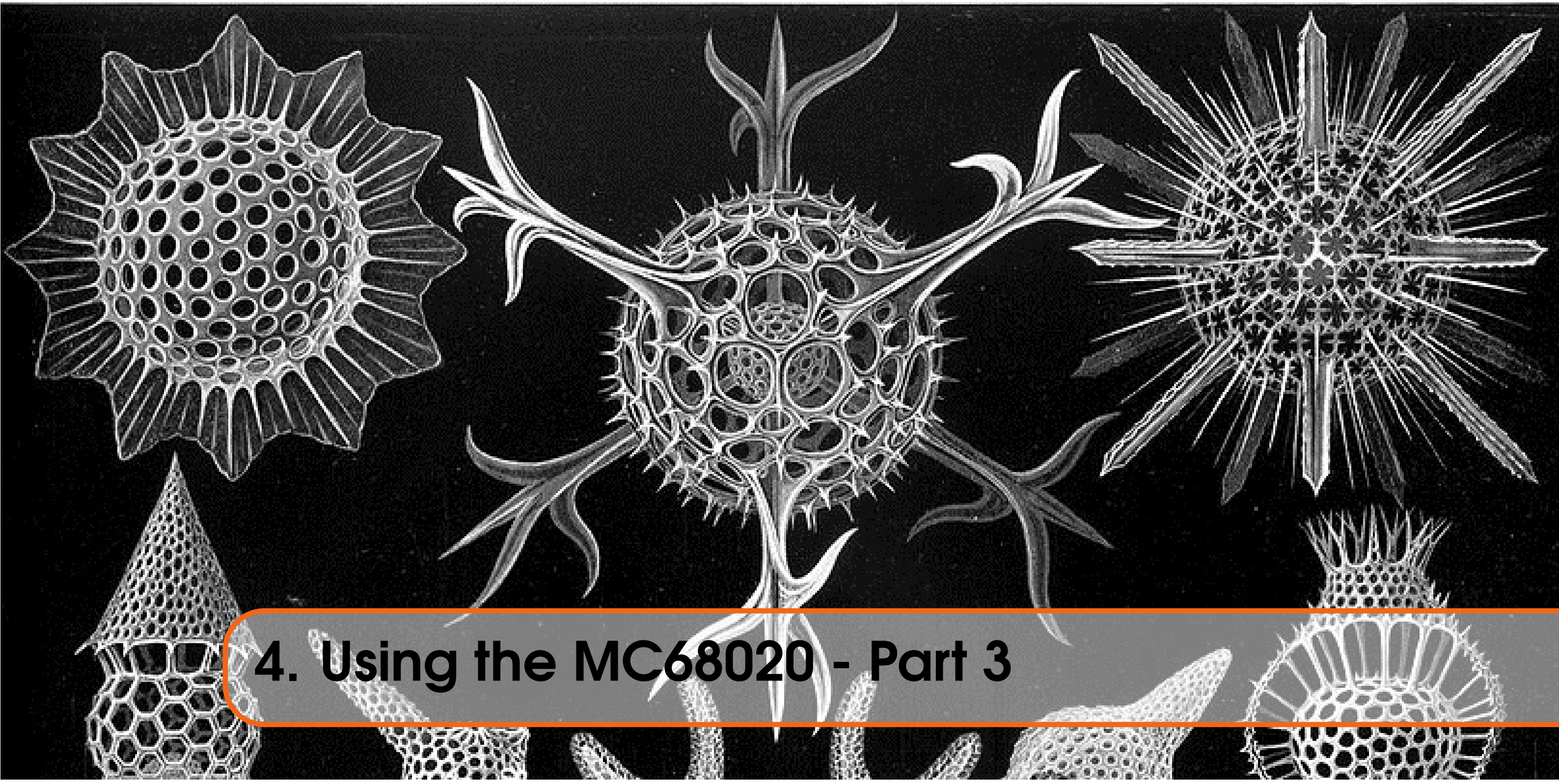
We can now write the file header back to the medium. This will set the data space and make the file executable.

```

;------------------------------------------------------------------------
; No errors, exit quietly back to SuperBASIC.
;------------------------------------------------------------------------
allDone
                moveq    #0,d0

;------------------------------------------------------------------------
; We have hit an error so we copy the code to D3 then exit via a
; forcible removal of this job. EXEC_W/EW will display the error in
; SuperBASIC, but EXEC/EX will not.
;------------------------------------------------------------------------
errorExit
                move.l   d0,d3                 ; Error code we want to return

;------------------------------------------------------------------------
; Kill myself when an error was detected, or at EOF.
;------------------------------------------------------------------------
suicide
                moveq    #mt_frjob,d0          ; This job will die soon
                moveq    #me,d1
                trap     #1
```

Listing 3.11: XTcc - Termination

The end. This is where we exit from the utility either with an error code or not.

Be aware that you will only ever see the error code or message, when you call the utility with EW as EX will not hang around to find out what the error, if any, was - it creates the job, activates it, and bales out. Only EW hangs around to the bitter end!



4. Using the MC68020 - Part 3

In the last issue, we took a very long look at the new and upgraded instructions that are now available when using an MC68020 processor as found in QPC - and possibly, in other emulators too. The old BBQL[1] uses an MC68008 and cannot cope with the new stuff.

To assemble these 62020 instructions, you need a copy of Gwass available from George's web site.[2]

This article continues our look at new features of the MC68020.

Here are the subjects I will cover in this issue, in relation to the 68020:

- The new format Status Register
- The various Control Registers used by the `MOVEC` instruction.

4.1 Status Register

The status register looks like the following in the MV68020:

Bit															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
T_1	T_0	S	M	-	I_2	I_1	I_0	-	-	-	X	N	Z	V	C

Table 4.1: MC68020 Status Register

4.1.1 Trace Bits T_1 and T_0

In the status register for the MC68020 we have now got an extra Trace bit - bit 14 - known as T_0. The original (MC68008) Trace bit, bit 15, is now known as the T_1 bit. Between the two Trace bits,

[1] Black Box QL

[2] http://gwiltprogs.info/page2.htm

better tracing can take place, as follows:

- 00 - When both Trace bits are zero, no tracing takes place.
- 01 - When T_1 is clear and T_0 is set, tracing takes place on a change of program flow - a branch, jump or subroutine call.
- 10 - When T_1 is set and T_0 is clear, tracing happens after every instruction. This is the tracing mode we are used to on the MC68008.
- 11 - Undefined. Probably best avoided!

4.1.2 Supervisor Master and Interrupt Modes

In addition to the extra Trace bit, there is a new Master bit as well. Bit 12 is the new Master bit.

On the MC68020, Supervisor mode is now split into two sub modes - master and interrupt. When the S and M bits are set then the processor is running in Master mode and uses the new Master Stack with the Master Stack Pointer in A7. (MSP(A7”))

When the S bit is set, and the M bit is clear, then the processor is running in Interrupt mode and uses another new stack, the Interrupt Stack, with A7 being the Interrupt Stack Pointer. (ISP(A7’))

The only difference between the two modes is the different stack pointer in use in register `A7`.

4.2 Control Registers and `MOVEC`

On the MC68020 we have the following control registers:

Control Register	Description
SFC	Source Function Code
DFC	Destination Function Code
USP	User Stack Pointer
VBR	Vector Base Register
CACR	Cache Control Register
CAAR	Cache Address Register
MSP	Master Stack Pointer
ISP	Interrupt Stack Pointer

Table 4.2: MC68020 Control Registers

4.2.1 SFC and DFC- Source and Destination Function Code

The alternate function code registers contain 3-bit function codes. Function codes can be considered extensions of the 32-bit logical address that optionally provides as many as eight 4-Gbyte address spaces - potentially increasing the 32 bit address bus to 35 bits.

The processor automatically generates function codes to select address spaces for data and programs at the user and supervisor modes.

Certain instructions use SFC and DFC to specify the function codes for operations.

The processor has three pins named FC0, FC1 and FC2. When the processor reads or writes from memory, these pins reflect information about the state of the processor.

They show the state of the processor - is it running in user or supervisor mode - and whether it is accessing data or instructions in memory.

The function codes are often used by external Memory Management Units (MMU) to protect various sections of memory. To the best of my knowledge, the QL doesn't have an MMU.

4.2.2 VBR - vector Base Register

The VBR is a 32 bit register which contains the base address of the exception vector table in memory. The displacement of an exception vector adds to the value in this register, which accesses the vector table.

On the MC68008, the exception table always lived at address 0, however, from the MC68010 onwards, the vector table still lives at address 0, but after a processor reset, the VBR can be adjusted to any desired location - provided that it can be addressed by a single 32 bit register.

4.2.3 CACR and CAAR - Cache Control

Many programs spend a lot of time executing loops. While within these loops, they execute the same (small) set of instructions over and over again. Each time the processor needs to execute an instruction, it must read it from memory.

There is a 256 byte instruction cache built in to the MC68020 (but probably not built in to the virtual MC68020 using in QPC, for example) which contains the most recently executed instructions.

In the case of a loop, the processor doesn't need to access memory to read the instructions more than once, in theory. When an instruction is read, it is stored in the cache and if executed again, will be read from cache which is much much quicker than reading from memory.

This is not always appropriate though, so the processor has the ability to enable, disable and otherwise manipulate the cache through the use of the CACR and CAAR control registers. These registers are 32 bits wide.

The use of these registers is beyond the scope of this series. They are unlikely to be mentioned ever again - except in passing, maybe!

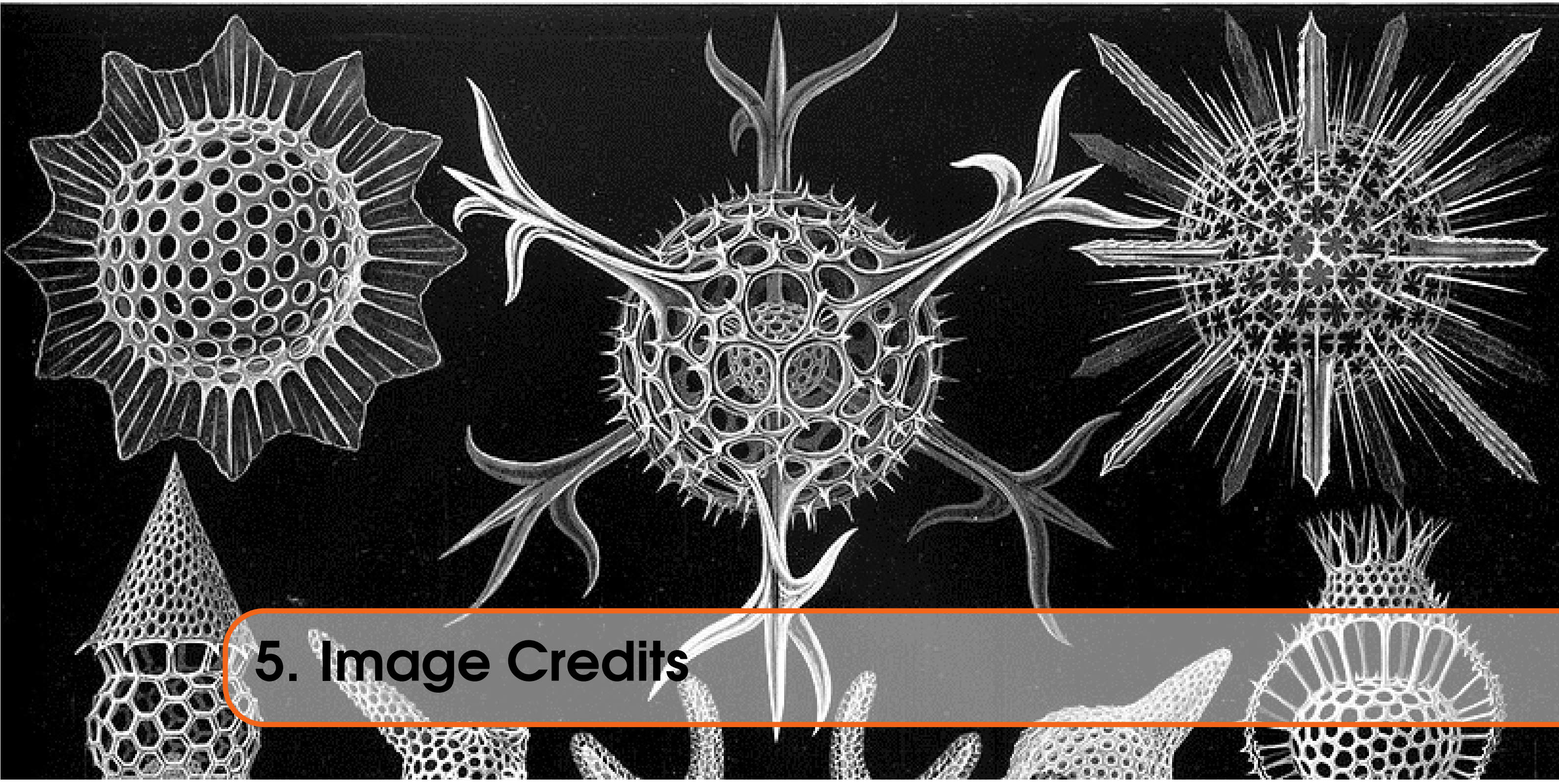
4.2.4 USP, MSP and ISP - Stack Pointers

In normal user programs, the processor runs in user mode and the stack pointer in `A7` is the USP or User Stack Pointer.

In Supervisor mode, a different stack is in use, usually limited in size, and on the BBQL, `A7` was then known as the SSP or Supervisor Stack Pointer.

On the MC68020 we have two submodes for Supervisor mode, and each one can have a different stack area and `A7` will be set accordingly to the Master Stack Pointer (MSP) or the Interrupt Stack Pointer (ISP) depending on the settings of the S and M bits in the Status Register.

If S is set and M is clear, the ISP is in `A7`, while the MSP is in `A7` if both bits are set.



5. Image Credits

The front cover image on this ePeriodical is taken from the book *Kunstformen der Natur* by German biologist Ernst Haeckel. The book was published between 1899 and 1904. The image used is of various *Polycystines* which are a specific kind of micro-fossil.

I have also cropped the image for use on each chapter heading page.

You can read about Polycystines on Wikipedia and there is a brief overview of the above book, also on Wikipedia, which shows a number of other images taken from the book. (Some of which I considered before choosing the current one!)

Polycystines have absolutely nothing to do with the QL or computing in general - in fact, I suspect they died out before electricity was invented - but I liked the image, and decided that it would make a good cover for the book and a decent enough chapter heading image too.

Not that I am suggesting, *in any way whatsoever*, that we QL fans are ancient.